Department of Computer Science and Engineering

National Institute of Technology Srinagar

Hazratbal, Srinagar, Jammu and Kashmir - 190006, India

LAB RECORD

Course Name: Compiler Design [CST416]

Submitted by

Niyati Gupta

2020BCSE073

IV Year B.Tech. CSE (7th Semester)

Submitted to

DR TAUSEEF AYOUB SHAIKH

Department of Computer Science and Engineering

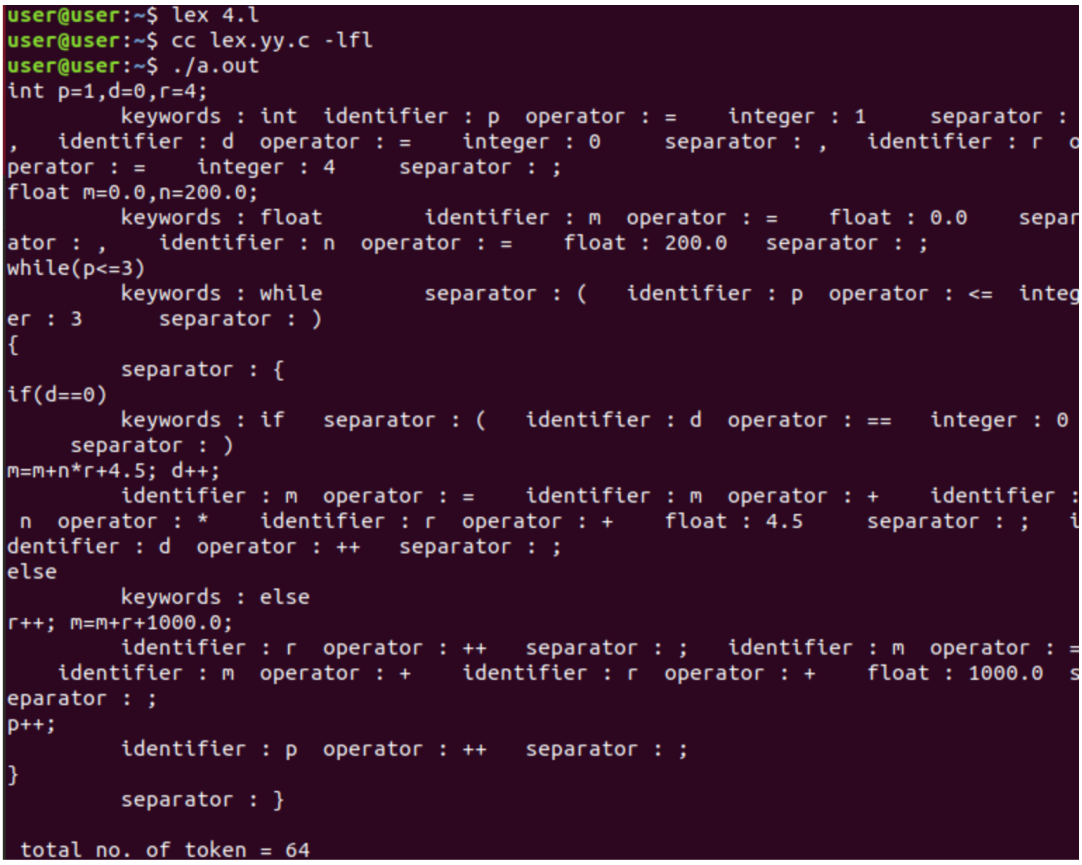
National Institute of Technology Srinagar

Hazratbal, Srinagar, Jammu and Kashmir – 190006, India

AUTUMN 2023

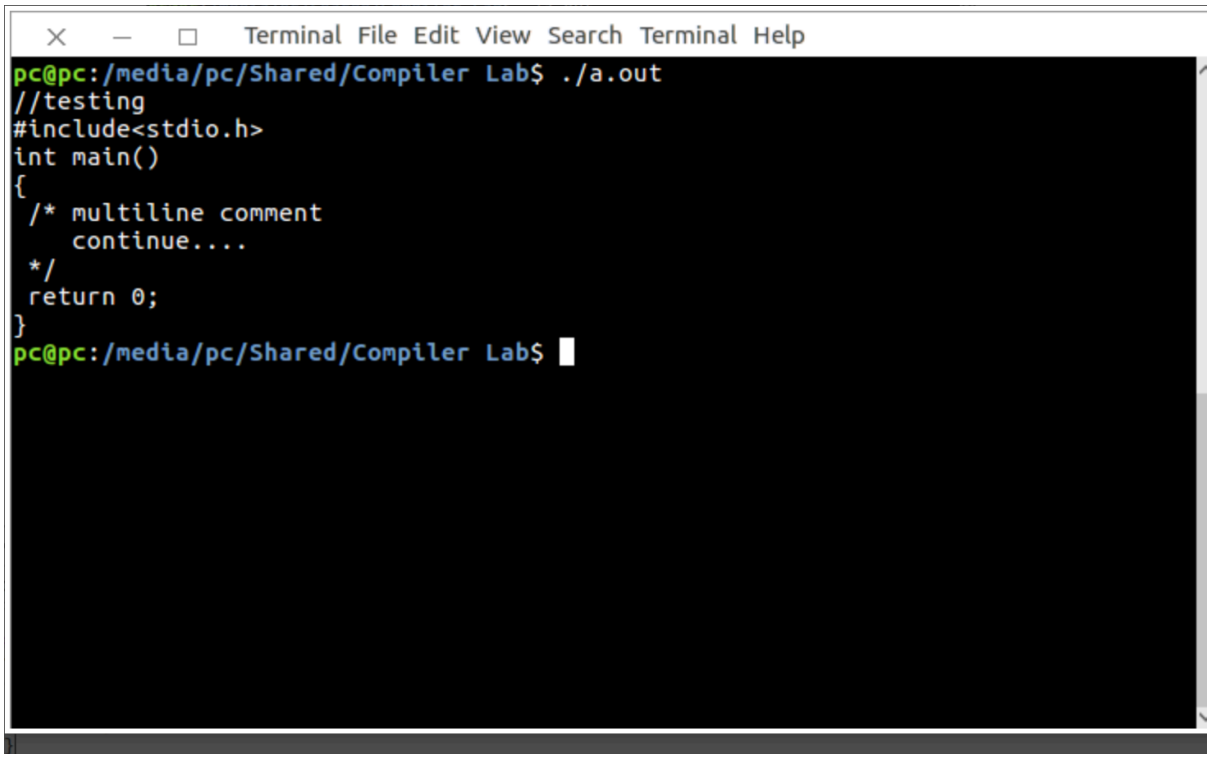
INDEX

OUTPUT :



```
user@user:~$ lex 4.l
user@user:~$ cc lex.yy.c -lfl
user@user:~$ ./a.out
int p=1,d=0,r=4;
        keywords : int  identifier : p  operator : =    integer : 1     separator :
,   identifier : d  operator : =    integer : 0     separator : ,   identifier : r  o
perator : =    integer : 4     separator : ;
float m=0.0,n=200.0;
        keywords : float        identifier : m  operator : =    float : 0.0    separ
ator : ,    identifier : n  operator : =    float : 200.0   separator : ;
while(p<=3)
        keywords : while        separator : (   identifier : p  operator : <=  integ
er : 3      separator : )
{
        separator : {
if(d==0)
        keywords : if   separator : (   identifier : d  operator : ==   integer : 0
     separator : )
m=m+n*r+4.5; d++;
        identifier : m  operator : =    identifier : m  operator : +    identifier :
 n  operator : *    identifier : r  operator : +    float : 4.5     separator : ;   i
dentifier : d  operator : ++   separator : ;
else
        keywords : else
r++; m=m+r+1000.0;
        identifier : r  operator : ++   separator : ;   identifier : m  operator : =
    identifier : m  operator : +    identifier : r  operator : +    float : 1000.0  s
eparator : ;
p++;
        identifier : p  operator : ++   separator : ;
}
        separator : }

 total no. of token = 64
```

OUTPUT :


Terminal File Edit View Search Terminal Help
pc@pc:/media/pc/Shared/Compiler Lab$ ./a.out
//testing
#include<stdio.h>
int main()
{
/* multiline comment
continue....
*/
return 0;
}
pc@pc:/media/pc/Shared/Compiler Lab$

LAB: 03

Aim: Write a program to convert Non-Deterministic Grammar to Deterministic Grammar

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LEN 100
char NFA_FILE[MAX_LEN];
char buffer[MAX_LEN];
int zz = 0;
// Structure to store DFA states and their
// status ( i.e new entry or already present)
struct DFA {
  char *states;
  int count;
} dfa;
int last_index = 0;
FILE *fp;
int symbols;
/* reset the hash map*/
void reset(int ar[], int size) {
  int i;
  // reset all the values of
  // the mapping array to zero
  for (i = 0; i < size; i++) {
    ar[i] = 0;
  }
}
// Check which States are present in the e-closure
```

```
/* map the states of NFA to a hash set*/
void check(int ar[], char S[]) {
  int i, j;
  // To parse the individual states of NFA
  int len = strlen(S);
  for (i = 0; i < len; i++) {
    // Set hash map for the position
    // of the states which is found
    j = ((int)(S[i]) - 65);
    ar[j]++;
  }
}
// To find new Closure States
void state(int ar[], int size, char S[]) {
  int j, k = 0;

  // Combine multiple states of NFA
  // to create new states of DFA
  for (j = 0; j < size; j++) {
    if (ar[j] != 0)
      S[k++] = (char)(65 + j);
  }
  // mark the end of the state
  S[k] = '\0';
}
// To pick the next closure from closure set
int closure(int ar[], int size) {
  int i;
  // check new closure is present or not
  for (i = 0; i < size; i++) {
    if (ar[i] == 1)
      return i;
```

```
  }
  return (100);
}
// Check new DFA states can be
// entered in DFA table or not
int indexing(struct DFA *dfa) {
  int i;
  for (i = 0; i < last_index; i++) {
    if (dfa[i].count == 0)
      return 1;
  }
  return -1;
}
/* To Display epsilon closure*/
void Display_closure(int states, int closure_ar[],
                     char *closure_table[],
                     char *NFA_TABLE[][symbols + 1],
                     char *DFA_TABLE[][symbols]) {
  int i;
  for (i = 0; i < states; i++) {
    reset(closure_ar, states);
    closure_ar[i] = 2;

    // to neglect blank entry
    if (strcmp(&NFA_TABLE[i][symbols], "-") != 0) {
  // copy the NFA transition state to buffer
      strcpy(buffer, &NFA_TABLE[i][symbols]);
      check(closure_ar, buffer);
      int z = closure(closure_ar, states);
      // till closure get completely saturated
      while (z != 100)
      {
```

```
      if (strcmp(&NFA_TABLE[z][symbols], "-") != 0) {
        strcpy(buffer, &NFA_TABLE[z][symbols]);

        // call the check function
        check(closure_ar, buffer);
      }
      closure_ar[z]++;
      z = closure(closure_ar, states);
    }
  }
  // print the e closure for every states of NFA
  printf("\n e-Closure (%c) :\t", (char)(65 + i));
  bzero((void *)buffer, MAX_LEN);
  state(closure_ar, states, buffer);
  strcpy(&closure_table[i], buffer);
  printf("%s\n", &closure_table[i]);
 }
}

/* To check New States in DFA */
int new_states(struct DFA *dfa, char S[]) {
 int i;

 // To check the current state is already
 // being used as a DFA state or not in
 // DFA transition table
 for (i = 0; i < last_index; i++) {
  if (strcmp(&dfa[i].states, S) == 0)
   return 0;
 }
 // push the new
 strcpy(&dfa[last_index++].states, S);
```

```
  // set the count for new states entered
  // to zero
  dfa[last_index - 1].count = 0;
  return 1;
}
// Transition function from NFA to DFA
// (generally union of closure operation )
void trans(char S[], int M, char *clsr_t[], int st,
            char *NFT[][symbols + 1], char TB[]) {
  int len = strlen(S);
  int i, j, k, g;
  int arr[st];
  int sz;
  reset(arr, st);
  char temp[MAX_LEN], temp2[MAX_LEN];
  char *buff;
  // Transition function from NFA to DFA
  for (i = 0; i < len; i++) {
    j = ((int)(S[i] - 65));
    strcpy(temp, &NFT[j][M]);
if (strcmp(temp, "-") != 0) {
      sz = strlen(temp);
      g = 0;
      while (g < sz) {
        k = ((int)(temp[g] - 65));
        strcpy(temp2, &clsr_t[k]);
        check(arr, temp2);
        g++;
      }
    }
  }
  bzero((void *)temp, MAX_LEN);
```

```
  state(arr, st, temp);
  if (temp[0] != '\0') {
    strcpy(TB, temp);
  } else
    strcpy(TB, "-");
}
/* Display DFA transition state table*/
void Display_DFA(int last_index, struct DFA *dfa_states,
                 char *DFA_TABLE[][symbols]) {
  int i, j;
  printf("\n\n*************************************************************\n\n");
  printf("\t\t DFA TRANSITION STATE TABLE \t\t \n\n");
  printf("\n STATES OF DFA :\t\t");
  for (i = 1; i < last_index; i++)
    printf("%s, ", &dfa_states[i].states);
  printf("\n");
  printf("\n GIVEN SYMBOLS FOR DFA: \t");
  for (i = 0; i < symbols; i++)
    printf("%d, ", i);
  printf("\n\n");
  printf("STATES\t");
  for (i = 0; i < symbols; i++)
    printf("|%d\t", i);
  printf("\n");
  // display the DFA transition state table
  printf("--------+----------------------\n");
  for (i = 0; i < zz; i++) {
    printf("%s\t", &dfa_states[i + 1].states);
    for (j = 0; j < symbols; j++) {
      printf("|%s \t", &DFA_TABLE[i][j]);
    }
    printf("\n");
```

```
  }
}
// Driver Code
int main() {
  int i, j, states;
  char T_buf[MAX_LEN];
  // creating an array dfa structures
  struct DFA *dfa_states = malloc(MAX_LEN * (sizeof(dfa)));
  states = 6, symbols = 2;
  printf("\n STATES OF NFA :\t\t");
  for (i = 0; i < states; i++)
    printf("%c, ", (char)(65 + i));
  printf("\n");
  printf("\n GIVEN SYMBOLS FOR NFA: \t");

  for (i = 0; i < symbols; i++)
    printf("%d, ", i);
  printf("eps");
  printf("\n\n");
  char *NFA_TABLE[states][symbols + 1];
  // Hard coded input for NFA table
  char *DFA_TABLE[MAX_LEN][symbols];
  strcpy(&NFA_TABLE[0][0], "FC");
  strcpy(&NFA_TABLE[0][1], "-");
  strcpy(&NFA_TABLE[0][2], "BF");
  strcpy(&NFA_TABLE[1][0], "-");
  strcpy(&NFA_TABLE[1][1], "C");
  strcpy(&NFA_TABLE[1][2], "-");
  strcpy(&NFA_TABLE[2][0], "-");
  strcpy(&NFA_TABLE[2][1], "-");
  strcpy(&NFA_TABLE[2][2], "D");
  strcpy(&NFA_TABLE[3][0], "E");
```

```
strcpy(&NFA_TABLE[3][1], "A");
strcpy(&NFA_TABLE[3][2], "-");
strcpy(&NFA_TABLE[4][0], "A");
strcpy(&NFA_TABLE[4][1], "-");
strcpy(&NFA_TABLE[4][2], "BF");
strcpy(&NFA_TABLE[5][0], "-");
strcpy(&NFA_TABLE[5][1], "-");
strcpy(&NFA_TABLE[5][2], "-");
printf("\n NFA STATE TRANSITION TABLE \n\n\n");
printf("STATES\t");
for (i = 0; i < symbols; i++)
  printf("|%d\t", i);
printf("eps\n");
// Displaying the matrix of NFA transition table
printf("--------+-----------------------------------\n");
for (i = 0; i < states; i++) {
  printf("%c\t", (char)(65 + i));
  for (j = 0; j <= symbols; j++) {
    printf("|%s \t", &NFA_TABLE[i][j]);
  }
  printf("\n");
}
int closure_ar[states];
char *closure_table[states];
Display_closure(states, closure_ar, closure_table, NFA_TABLE, DFA_TABLE);
strcpy(&dfa_states[last_index++].states, "-");
dfa_states[last_index - 1].count = 1;
bzero((void *)buffer, MAX_LEN);
strcpy(buffer, &closure_table[0]);
strcpy(&dfa_states[last_index++].states, buffer);
int Sm = 1, ind = 1;
int start_index = 1;
```

```
 // Filling up the DFA table with transition values
 // Till new states can be entered in DFA table
 while (ind != -1) {
   dfa_states[start_index].count = 1;
   Sm = 0;
   for (i = 0; i < symbols; i++) {
     trans(buffer, i, closure_table, states, NFA_TABLE, T_buf);
// storing the new DFA state in buffer
strcpy(&DFA_TABLE[zz][i], T_buf);
     // parameter to control new states
     Sm = Sm + new_states(dfa_states, T_buf);
   }
   ind = indexing(dfa_states);
   if (ind != -1)
     strcpy(buffer, &dfa_states[++start_index].states);
   zz++;
 }
 // display the DFA TABLE
 Display_DFA(last_index, dfa_states, DFA_TABLE);

 return 0;
}
```

OUTPUT :

```
 STATES OF NFA :                A, B, C, D, E, F,

 GIVEN SYMBOLS FOR NFA:         0, 1, eps


 NFA STATE TRANSITION TABLE


STATES  |0      |1      eps
--------+-------------------------------------
A       |FC     |-      |BF
B       |-      |C      |-
C       |-      |-      |D
D       |E      |A      |-
E       |A      |-      |BF
F       |-      |-      |-

 e-Closure (A) :        ABF

 e-Closure (B) :        B

 e-Closure (C) :        CD

 e-Closure (D) :        D

 e-Closure (E) :        BEF

 e-Closure (F) :        F


**********************************************************

                DFA TRANSITION STATE TABLE


 STATES OF DFA :                ABF, CDF, CD, BEF,

 GIVEN SYMBOLS FOR DFA:         0, 1,

STATES  |0      |1
--------+----------------------
ABF     |CDF    |CD
CDF     |BEF    |ABF
CD      |BEF    |ABF
BEF     |ABF    |CD
(base) nivatiqupta@Maccy 2020BCSE073 %
```

LAB: 04

Aim: Write a program to calculate First and Follow of a given LL(1) grammar.

Code:

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);
// Function to calculate First
void findfirst(char, int, int);
int count, n = 0;
// Stores the final result
// of the First Sets
char calc_first[10][100];
// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;
// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;
int main(int argc, char** argv)
{
   int jm = 0;
   int km = 0;
   int i, choice;
   char c, ch;
   count = 8;
```

```
    // The Input grammar
    strcpy(production[0], "X=TnS");
    strcpy(production[1], "X=Rm");
    strcpy(production[2], "T=q");
    strcpy(production[3], "T=#");
    strcpy(production[4], "S=p");
    strcpy(production[5], "S=#");
    strcpy(production[6], "R=om");
    strcpy(production[7], "R=ST");
    int kay;
    char done[count];
    int ptr = -1;
    // Initializing the calc_first array
    for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
            calc_first[k][kay] = '!';
        }
    }
    int point1 = 0, point2, xxx;
    for (k = 0; k < count; k++) {
        c = production[k][0];
        point2 = 0;
        xxx = 0;
        // Checking if First of c has
        // already been calculated
        for (kay = 0; kay <= ptr; kay++)
if (c == done[kay])
                xxx = 1;
        if (xxx == 1)
            continue;
        // Function call
        findfirst(c, 0, 0);
        ptr += 1;
```

```
    // Adding c to the calculated list
    done[ptr] = c;
    printf("\n First(%c) = { ", c);
    calc_first[point1][point2++] = c;
    // Printing the First Sets of the grammar
    for (i = 0 + jm; i < n; i++) {
      int lark = 0, chk = 0;
      for (lark = 0; lark < point2; lark++) {
        if (first[i] == calc_first[point1][lark]) {
          chk = 1;
          break;
        }
      }
      if (chk == 0) {
        printf("%c, ", first[i]);
        calc_first[point1][point2++] = first[i];
      }
    }
    printf("}\n");
    jm = n;
    point1++;
  }
printf("\n");
 printf("-----------------------------------------------"
      "\n\n");
 char donee[count];
 ptr = -1;
 // Initializing the calc_follow array
 for (k = 0; k < count; k++) {
    for (kay = 0; kay < 100; kay++) {
      calc_follow[k][kay] = '!';
    }
 }
```

```
point1 = 0;
int land = 0;
for (e = 0; e < count; e++) {
   ck = production[e][0];
   point2 = 0;
   xxx = 0;
   // Checking if Follow of ck
   // has already been calculated
   for (kay = 0; kay <= ptr; kay++)
      if (ck == donee[kay])
         xxx = 1;
   if (xxx == 1)
      continue;
   land += 1;
   // Function call
   follow(ck);
   ptr += 1;
   // Adding ck to the calculated list
   donee[ptr] = ck;
   printf(" Follow(%c) = { ", ck);
   calc_follow[point1][point2++] = ck;
   // Printing the Follow Sets of the grammar
   for (i = 0 + km; i < m; i++) {
      int lark = 0, chk = 0;
      for (lark = 0; lark < point2; lark++) {
         if (f[i] == calc_follow[point1][lark]) {
            chk = 1;
            break;
         }
      }
      if (chk == 0) {
         printf("%c, ", f[i]);
         calc_follow[point1][point2++] = f[i];
```

```
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
}
void follow(char c)
{
    int i, j;
    // Adding "$" to the follow
    // set of the start symbol
    if (production[0][0] == c) {
        f[m++] = '$'; }
    for (i = 0; i < 10; i++) {
        for (j = 2; j < 10; j++) {
            if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                    // Calculate the first of the next
                    // Non-Terminal in the production
                    followfirst(production[i][j + 1], i,
                                (j + 2));
                }
                if (production[i][j + 1] == '\0'
                    && c != production[i][0]) {
                    // Calculate the follow of the
                    // Non-Terminal in the L.H.S. of the
                    // production
                    follow(production[i][0]);
                }
            }
        }
    }
```

```
}
void findfirst(char c, int q1, int q2)
{
    int j;
    // The case where we
    // encounter a Terminal
    if (!(isupper(c))) {
        first[n++] = c;
    }
    for (j = 0; j < count; j++) {
        if (production[j][0] == c) {
            if (production[j][2] == '#') {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0'
                        && (q1 != 0 || q2 != 0)) {
                    // Recursion to calculate First of New
                    // Non-Terminal we encounter after
                    // epsilon
                    findfirst(production[q1][q2], q1,
                            (q2 + 1));
                }
                else
                    first[n++] = '#';
            }
            else if (!isupper(production[j][2])) {
                first[n++] = production[j][2];
            }
            else {
                // Recursion to calculate First of
                // New Non-Terminal we encounter
                // at the beginning
                findfirst(production[j][2], j, 3);
```

```
        }
      }
   }
}
void followfirst(char c, int c1, int c2)
{   int k;
   // The case where we encounter
   // a Terminal
   if (!(isupper(c)))
      f[m++] = c;
   else {
      int i = 0, j = 1;
      for (i = 0; i < count; i++) {
         if (calc_first[i][0] == c)
            break;
      }       // Including the First set of the
      // Non-Terminal in the Follow of
      // the original query
      while (calc_first[i][j] != '!') {
         if (calc_first[i][j] != '#') {
            f[m++] = calc_first[i][j];
         }
         else {
            if (production[c1][c2] == '\0') {
               // Case where we reach the
               // end of a production
               follow(production[c1][0]);
            } else {
               // Recursion to the next symbol
               // in case we encounter a "#"
               followfirst(production[c1][c2], c1,
                           c2 + 1);
            }   }j++;}}}
```

Output:

```
(base) niyatigupta@Maccy 2020BCSE073 % cd "/Users/niyatigupta/Desktop/2020BCSE073/" && gcc b.c -o b && "/Users/niyatigupta/Desktop/2020BCSE073/"b
 First(X) = { q, n, o, p, #, }
 First(T) = { q, #, }
 First(S) = { p, #, }
 First(R) = { o, p, q, #, }
--------------------------------------------
 Follow(X) = { $,  }
 Follow(T) = { n, m,  }
 Follow(S) = { $, q, m,  }
 Follow(R) = { m,  }
(base) niyatigupta@Maccy 2020BCSE073 %
```

LAB: 05

Aim: Write a program to construct LL(1) parsing table for LL(1) grammar and validate the input string .

Code:

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
void followfirst(char , int , int);
void findfirst(char , int , int);
void follow(char c);
int count,n=0;
char calc_first[10][100];
char calc_follow[10][100];
int m=0;
char production[10][10], first[10];
char f[10];
int k;
char ck;
int e;
int main(int argc,char **argv){
        int jm =0;
        int km=0;
        int i,choice;
        char c,ch;
        printf("How many productions ? :");
        scanf("%d",&count);
        printf("\nEnter %d productions in form A=B where A and B are grammar symbols :
\n\n",count);
        for(i=0;i<count;i++){
                scanf("%s%c",production[i],&ch);
        }
```

```
int kay;
char done[count];
int ptr = -1;
for(k=0;k<count;k++){
        for(kay=0;kay<100;kay++){
                calc_first[k][kay] = '!';
        }
}
int point1 = 0,point2,xxx;
for(k=0;k<count;k++){
        c=production[k][0];
        point2 = 0;
        xxx = 0;
        for(kay = 0; kay <= ptr; kay++)
                if(c == done[kay])          xxx = 1;
        if (xxx == 1)       continue;
        findfirst(c,0,0) ;
        ptr+=1 ;
        done[ptr] = c;
        printf("\n First(%c)= { ",c);
        calc_first[point1][point2++] = c;
        for(i=0+jm;i<n;i++){
                int lark = 0,chk = 0;
                for(lark=0;lark<point2;lark++){
                        if (first[i] == calc_first[point1][lark]){
                                chk = 1,  break;
                        }
                }
                if(chk == 0){
                        printf("%c, ",first[i]);
                        calc_first[point1][point2++] = first[i];
                }}
```

```
            printf("}\n");
            jm=n;
            point1++;}
printf("\n");
printf("----------------------------------------------\n\n");
char donee[count];
ptr = -1;
for(k=0;k<count;k++){
            for(kay=0;kay<100;kay++){
                        calc_follow[k][kay] = '!';
            }
}           point1 = 0;
int land = 0 ;
for(e=0;e<count;e++){
            ck=production[e][0];
            point2 = 0;
            xxx = 0;
            for(kay = 0; kay <= ptr; kay++)
                        if(ck == donee[kay])       xxx = 1;
            if (xxx == 1)       continue;
            land += 1;
            follow(ck);
            ptr+=1;
            donee[ptr] = ck;
            printf(" Follow(%c) = { ",ck);
            calc_follow[point1][point2++] = ck;
            for(i=0+km;i<m;i++){
                        int lark = 0,chk = 0;
                        for(lark=0;lark<point2;lark++){
                                    if (f[i] == calc_follow[point1][lark]){
                                                chk = 1;
                                                break;
```

```
                        }
                    }
                    if(chk == 0){
                        printf("%c, ",f[i]);
                        calc_follow[point1][point2++] = f[i];
                    }
                }
                printf(" }\n\n");
                km=m;
                point1++;
            }
            char ter[10];
            for(k=0;k<10;k++){
                ter[k] = '!';
            }
            int ap,vp,sid = 0;
            for(k=0;k<count;k++){
                for(kay=0;kay<count;kay++){
                    if(!isupper(production[k][kay]) && production[k][kay]!= '#' && production[k]
[kay] != '=' && production[k][kay] != '\0'){
                        vp = 0;
                        for(ap = 0;ap < sid; ap++){
                            if(production[k][kay] == ter[ap]){
                                vp = 1;
                                break;
                            }
                        }
                        if(vp == 0){
                            ter[sid] = production[k][kay];
                            sid ++;
                        }
                    }
```

```
                }
        }
        ter[sid] = '$';
        sid++;
        printf("\n\t\t\t\t\t\t\t The LL(1) Parsing Table for the above grammer :-");
        printf("\n\t\t\t\t\t\t\t^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");

printf("\n\t\t\t==========================================================================
==========================================\n");
        printf("\t\t\t\t|\t");
        for(ap = 0;ap < sid; ap++){
                printf("%c\t\t",ter[ap]);
        }

printf("\n\t\t\t==========================================================================
==========================================\n");
        char first_prod[count][sid];
        for(ap=0;ap<count;ap++){
                int destiny = 0;
                k = 2;
                int ct = 0;
                char tem[100];
                while(production[ap][k] != '\0'){
                        if(!isupper(production[ap][k])){
                                tem[ct++] = production[ap][k];
                                tem[ct++] = '_';
                                tem[ct++] = '\0';
                                k++;
                                break;
                        }
                        else{
                                int zap=0;
                                int tuna = 0;
```

```
                        for(zap=0;zap<count;zap++){
                                if(calc_first[zap][0] == production[ap][k]){
                                        for(tuna=1;tuna<100;tuna++){
                                                if(calc_first[zap][tuna] != '!'){
                                                        tem[ct++] = calc_first[zap][tuna];
                                                }
                                                else    break;
                                        }       break;
                                }}
                        tem[ct++] = '_';
                }k++;
        }
        int zap = 0,tuna;
        for(tuna = 0;tuna<ct;tuna++){
                if(tem[tuna] == '#'){
                        zap = 1;
                }else if(tem[tuna] == '_'){
                        if(zap == 1)      zap = 0;
                        else    break;
                }else   first_prod[ap][destiny++] = tem[tuna];
        }
}       char table[land][sid+1];
ptr = -1 ;
for(ap = 0; ap < land ; ap++){
        for(kay = 0; kay < (sid + 1) ; kay++){
                table[ap][kay] = '!';
        }
}
for(ap = 0; ap < count ; ap++){
        ck = production[ap][0];
        xxx = 0;
        for(kay = 0; kay <= ptr; kay++)
```

```
                if(ck == table[kay][0])xxx = 1;
        if (xxx == 1)       continue;
        else{
                ptr = ptr + 1;
                table[ptr][0] = ck;
        }
}
for(ap = 0; ap < count ; ap++){
        int tuna = 0;
        while(first_prod[ap][tuna] != '\0'){
                int to,ni=0;
                for(to=0;to<sid;to++){
                        if(first_prod[ap][tuna] == ter[to])          ni = 1;
                }
                if(ni == 1){
                        char xz = production[ap][0];
                        int cz=0;
                        while(table[cz][0] != xz){
                                cz = cz + 1;
                        }
                        int vz=0;
                        while(ter[vz] != first_prod[ap][tuna]){
                                vz = vz + 1;
                        }
                        table[cz][vz+1] = (char)(ap + 65);
                }
                tuna++;
        }
}
for(k=0;k<sid;k++){
        for(kay=0;kay<100;kay++){
                if(calc_first[k][kay] == '!')          break ;
```

```
                else if(calc_first[k][kay] == '#'){
                        int fz = 1;
                        while(calc_follow[k][fz] != '!'){
                                char xz = production[k][0];
                                int cz=0;
                                while(table[cz][0] != xz){
                                        cz = cz + 1;
                                }
                                int vz=0;
                                while(ter[vz] != calc_follow[k][fz]){
                                        vz = vz + 1;
                                }
                                table[k][vz+1] = '#';
                                fz++;
                        }       break;
                }
        }
}
for(ap = 0; ap < land ; ap++){
        printf("\t\t\t   %c\t|\t",table[ap][0]);
        for(kay = 1; kay < (sid + 1) ; kay++){
                if(table[ap][kay] == '!')
                        printf("\t\t");
                else if(table[ap][kay] == '#')
                        printf("%c=#\t\t",table[ap][0]);
                else{
                        int mum = (int)(table[ap][kay]);
                        mum -= 65;
                        printf("%s\t\t",production[mum]);
                }
        }       printf("\n");
```

```
printf("\t\t\t----------------------------------------------------------------------------------------------------------
");
                printf("\n");
        }
        int j;
        printf("\n\nPlease enter the desired INPUT STRING = ");
        char input[100];
        scanf("%s%c",input,&ch);

printf("\n\t\t\t\t\t==========================================================================
========\n");
        printf("\t\t\t\t\t\tStack\t\t\tInput\t\t\tAction");

printf("\n\t\t\t\t\t==========================================================================
========\n");
        int i_ptr = 0,s_ptr = 1;
        char stack[100];
        stack[0] = '$';
        stack[1] = table[0][0];
        while(s_ptr != -1){
                printf("\t\t\t\t\t\t");
                int vamp = 0;
                for(vamp=0;vamp<=s_ptr;vamp++){
                        printf("%c",stack[vamp]);
                }
                printf("\t\t\t");
                vamp = i_ptr;
                while(input[vamp] != '\0'){
                        printf("%c",input[vamp]);
                        vamp++;
                }
                printf("\t\t\t");
                char her = input[i_ptr];
```

```
char him = stack[s_ptr];
s_ptr--;
if(!isupper(him)){
        if(her == him){
                i_ptr++;
                printf("POP ACTION\n");
        }
        else{
                printf("\nString Not Accepted by LL(1) Parser !!\n");
                exit(0);
        }
}else{
        for(i=0;i<sid;i++){
                if(ter[i] == her)
                        break;
        }
        char produ[100];
        for(j=0;j<land;j++){
                if(him == table[j][0]){
                        if (table[j][i+1] == '#'){
                                printf("%c=#\n",table[j][0]);
                                produ[0] = '#';
                                produ[1] = '\0';
                        }
                        else if(table[j][i+1] != '!'){
                                int mum = (int)(table[j][i+1]);
                                mum -= 65;
                                strcpy(produ,production[mum]);
                                printf("%s\n",produ);
                        }else{
                                printf("\nString Not Accepted by LL(1) Parser !!\n");
                                exit(0);
```

```
                                        }
                                }
                        }
                        int le = strlen(produ);
                        le = le - 1;
                        if(le == 0){
                                continue;
                        }
                        for(j=le;j>=2;j--){
                                s_ptr++;
                                stack[s_ptr] = produ[j];
                        }
                }
        }

printf("\n\t\t\t==========================================================================
================================================\n");
        if (input[i_ptr] == '\0'){
                printf("\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED !!\n");
        }else
                printf("\n\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN REJECTED !!\n");

printf("\t\t\t==========================================================================
================================================\n");
}
void follow(char c){
        int i ,j;
        if(production[0][0]==c){
                f[m++]='$';
        }
        for(i=0;i<10;i++){
                for(j=2;j<10;j++){
                        if(production[i][j]==c){
```

```
                        if(production[i][j+1]!='\0'){
                                followfirst(production[i][j+1],i,(j+2));
                        }if(production[i][j+1]=='\0'&&c!=production[i][0]){
                        follow(production[i][0]);
                        }
                        }
                }
        }
}
void findfirst(char c ,int q1 , int q2)
{
        int j;
        if(!(isupper(c)))
                first[n++]=c;
        for(j=0;j<count;j++){
                if(production[j][0]==c)
                {
                        if(production[j][2]=='#'){
                                if(production[q1][q2] == '\0')
                                        first[n++]='#';
                                else if(production[q1][q2] != '\0' && (q1 != 0 || q2 != 0))
                                {
                                        findfirst(production[q1][q2], q1, (q2+1));
                                }
                                else    first[n++]='#';
                        }
                        else if(!isupper(production[j][2]))
                                first[n++]=production[j][2] ;
                        else    findfirst(production[j][2], j, 3);
                }
        }
}
```

```
void followfirst(char c, int c1 , int c2){
   int k;
   if(!(isupper(c)))
                    f[m++]=c;
          else{
                    int i=0,j=1;
                    for(i=0;i<count;i++)
                    {
                              if(calc_first[i][0] == c)
                                        break;
                    }
                    while(calc_first[i][j] != '!')
                    {
                              if(calc_first[i][j] != '#'){
                                        f[m++] = calc_first[i][j];
                              }
                              else{
                                        if(production[c1][c2] == '\0')
                                                  follow(production[c1][0]);
                                        else      followfirst(production[c1][c2],c1,c2+1);
                              }
                              j++;
                    }
          }
}
```

Output:

```
(base) niyatigupta@Maccy 2020BCSE073 % cd "/Users/niyatigupta/Desktop/2020BCSE073/" && gcc d.c -o d && "/Users/niyatigupta/Desktop/2020BCSE073/"d
How many productions ? :8

Enter 8 productions in form A=B where A and B are grammar symbols :

E=TR
R=+TR
R=#
T=FY
Y=*FY
Y=#
F=(E)
F=i

 First(E)= { (, i, }

 First(R)= { +, #, }

 First(T)= { (, i, }

 First(Y)= { *, #, }

 First(F)= { (, i, }

------------------------------------------

 Follow(E) = { $, ),  }

 Follow(R) = { $, ),  }

 Follow(T) = { +, $, ),  }

 Follow(Y) = { +, $, ),  }

 Follow(F) = { *, +, $, ), (, i,  }

                                                The LL(1) Parsing Table for the above grammer :-

                        ================================================================================================================
                                |       +               $               (               i               *               )               $
                        ================================================================================================================
                           +    |                                       +$(i+
                        ----------------------------------------------------------------------------------------------------------------
                           R    |       R=+TR           R=#                                                             R=#
                        ----------------------------------------------------------------------------------------------------------------
                           T    |                                       T=FY            T=FY
                        ----------------------------------------------------------------------------------------------------------------
                           Y    |       Y=#             Y=#                                             Y=*FY           Y=#
                        ----------------------------------------------------------------------------------------------------------------
                           F    |                                       F=(E)           F=i
                        ----------------------------------------------------------------------------------------------------------------
```

```
 Please enter the desired INPUT STRING = i+i*i$

                                    ========================================================================
                                            Stack                   Input                   Action
                                    ========================================================================
                                            $+                      i+i*i$
 String Not Accepted by LL(1) Parser !!
                                            $                       i+i*i$
 String Not Accepted by LL(1) Parser !!

                        ================================================================================================================

                                                        YOUR STRING HAS BEEN REJECTED !!

                        ================================================================================================================
(base) niyatigupta@Maccy 2020BCSE073 %
```

LAB: 06

Aim: Write a program to construct operator precedence parsing table for the given grammar and check the validity of the input string (id+id*id).

E -> E+T | T

T-> T*F | F

F-> (E) | id

Code:

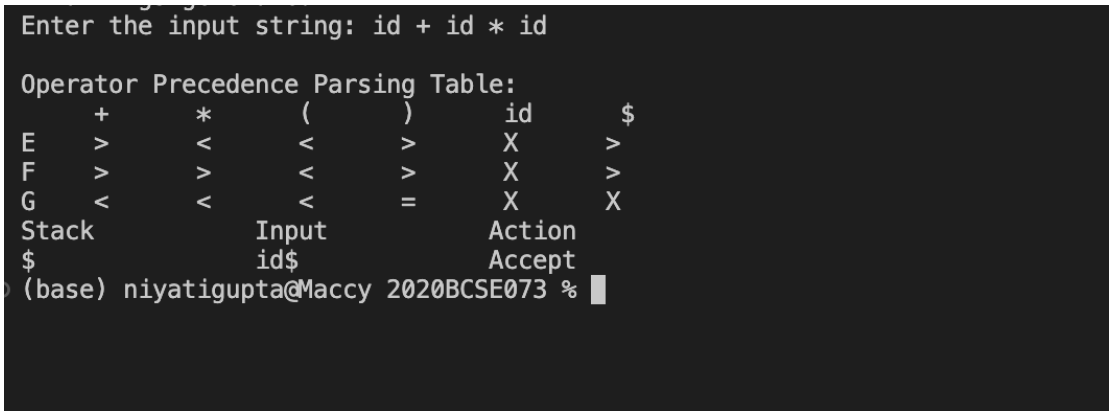
```
#include <stdio.h>
#include <string.h>
#define NUM_TERMINALS 6
#define NUM_NON_TERMINALS 3
#define MAX_LENGTH 50// Function to check if a character is an operator
int isOperator(char ch) {
   return (ch == '+' || ch == '*' || ch == '(' || ch == ')' || ch == 'id' || ch == '$');
}
// Function to get the precedence of an operator
char getPrecedence(char op) {
   switch (op) {
      case '+':
         return 1;
      case '*':
         return 2;
      case '(':
      case ')':
      case 'id':
      case '$':
         return 0;
      default:
         return -1; // Error
   }
}
```

```
// Function to initialize the operator precedence parsing table
void initializeTable(char table[][NUM_TERMINALS]) {
    // Terminals: +, *, (, ), id, $
    // Non-terminals: E, T, F
    // Initialize the table with X (no entry)
    for (int i = 0; i < NUM_NON_TERMINALS; i++) {
        for (int j = 0; j < NUM_TERMINALS; j++) {
            table[i][j] = 'X';
        }
    }
    // Fill in the entries based on the grammar
    table[0][0] = '>';  // E + T
    table[0][1] = '<';  // E * T
    table[0][2] = '<';  // E ( E )
    table[0][3] = '>';  // E id
    table[0][5] = '>';  // E $
    table[1][0] = '>';  // T + F
    table[1][1] = '>';  // T * F
    table[1][2] = '<';  // T ( E )
    table[1][3] = '>';  // T id
    table[1][5] = '>';  // T $
    table[2][0] = '<';  // F + T
    table[2][1] = '<';  // F * T
    table[2][2] = '<';  // F ( E )
    table[2][3] = '=';  // F id
    table[2][4] = 'X';  // F $
}

// Function to perform operator precedence parsing
int operatorPrecedenceParsing(char input[], char table[][NUM_TERMINALS]) {
    char stack[MAX_LENGTH];
```

```
int top = -1;

// Push a dollar sign onto the stack
stack[++top] = '$';

// Append a dollar sign to the end of the input
strcat(input, "$");

// Initialize the input pointer and stack symbol
int ip = 0;
char stackTop = stack[top];

printf("Stack\t\tInput\t\tAction\n");

while (stackTop != '$') {
   // Print the current stack and input
   printf("%-10s\t%-15s\t", stack, &input[ip]);

   // Check if the stack top and input symbol have the same precedence
   if (getPrecedence(stackTop) >= getPrecedence(input[ip])) {
      printf("Reduce ");
      // Simulate reducing based on precedence
      // This step is simplified for the given grammar
      top--;
   } else {
      printf("Shift ");
      // Simulate shifting the input symbol onto the stack
      stack[++top] = input[ip++];
   }

   // Move to the next line
   printf("\n");
```

```
        // Update the stack top
        stackTop = stack[top];
    }
    // Print the final action
    printf("%-10s\t%-15s\tAccept\n", stack, &input[ip]);
    return 0;
}
int main() {
    char input[MAX_LENGTH];
    char parsingTable[NUM_NON_TERMINALS][NUM_TERMINALS];
    // Input string: "id+id*id"
    printf("Enter the input string: ");
    scanf("%s", input);
    // Initialize and print the operator precedence parsing table
    initializeTable(parsingTable);
    printf("\nOperator Precedence Parsing Table:\n");
    printf("    +     *     (     )     id     $\n");
    for (int i = 0; i < NUM_NON_TERMINALS; i++) {
        printf("%c  ", 'E' + i);
        for (int j = 0; j < NUM_TERMINALS; j++) {
            printf("  %c    ", parsingTable[i][j]);
        }
        printf("\n");
    }
    // Perform operator precedence parsing
    operatorPrecedenceParsing(input, parsingTable);
    return 0;
}
```

Output:


Enter the input string: id + id * id
Operator Precedence Parsing Table:
+ * ( ) id $
E > < < > X >
F > > < > X >
G < < < = X X
Stack Input Action
$ id$ Accept
(base) niyatigupta@Maccy 2020BCSE073 %

LAB: 07

Aim: Write a program to implement recursive decent parser

Code:

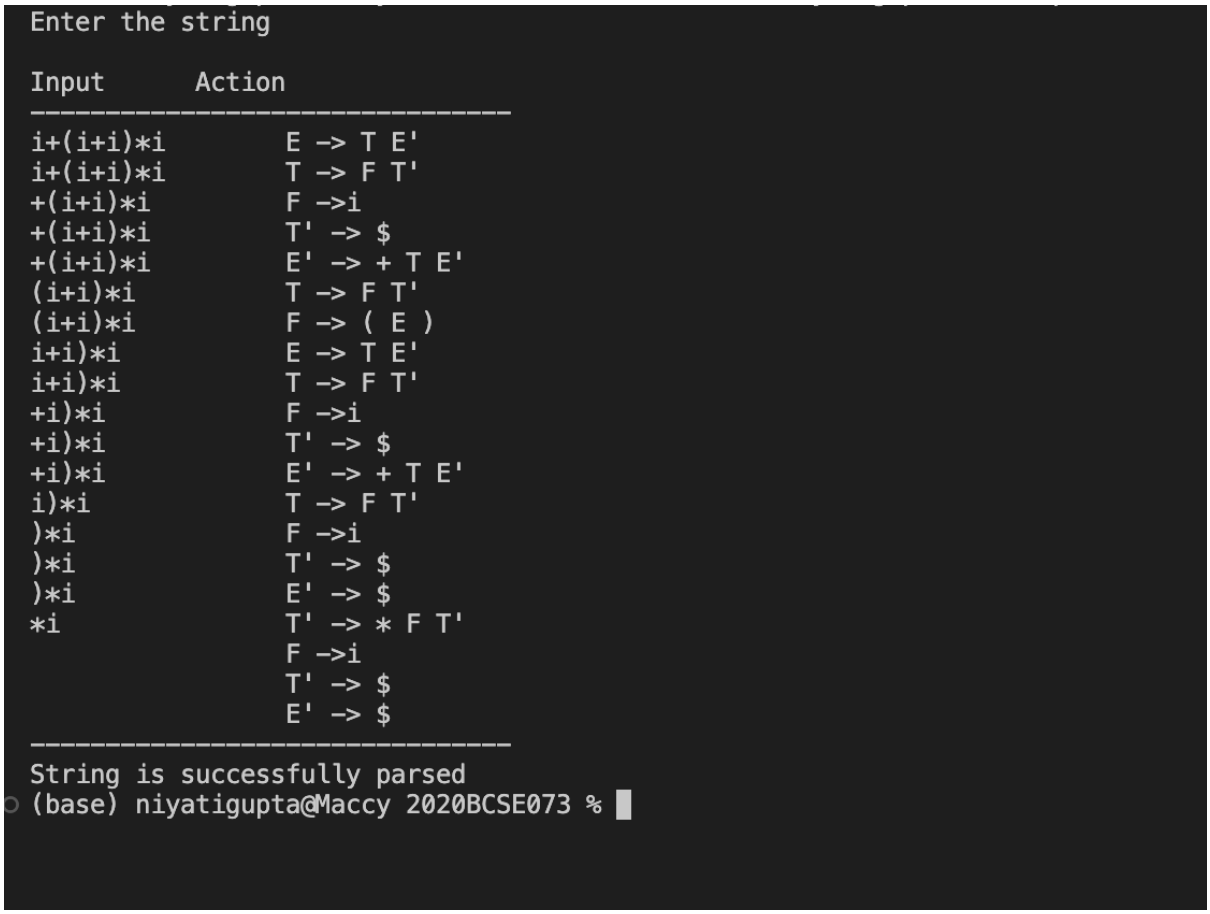
```
#include <stdio.h>
#include <string.h>
#define SUCCESS 1
#define FAILED 0
int E(), Edash(), T(), Tdash(), F();
const char *cursor;
char string[64];
int main()
{
   puts("Enter the string");
   // scanf("%s", string);
   sscanf("i+(i+i)*i", "%s", string);
   cursor = string;
   puts("");
   puts("Input      Action");
   puts("--------------------------------");
   if (E() && *cursor == '\0') {
      puts("--------------------------------");
      puts("String is successfully parsed");
      return 0;
   } else {
      puts("--------------------------------");
      puts("Error in parsing String");
      return 1;
   }
}

int E()
```

```
{
   printf("%-16s E -> T E'\n", cursor);
   if (T()) {
      if (Edash())
         return SUCCESS;
      else
         return FAILED;
   } else
      return FAILED;
}
int Edash()
{
   if (*cursor == '+') {
      printf("%-16s E' -> + T E'\n", cursor);
      cursor++;
      if (T()) {
         if (Edash())
            return SUCCESS;
         else
            return FAILED;
      } else
         return FAILED;
   } else {
      printf("%-16s E' -> $\n", cursor);
      return SUCCESS;
   }
}
int T()
{
   printf("%-16s T -> F T'\n", cursor);
   if (F()) {
      if (Tdash())
```

```
            return SUCCESS;
        else
            return FAILED;
    } else
        return FAILED;
}

int Tdash()
{
    if (*cursor == '*') {
        printf("%-16s T' -> * F T'\n", cursor);
        cursor++;
        if (F()) {
            if (Tdash())
                return SUCCESS;
            else
                return FAILED;
        } else
            return FAILED;
    } else {
        printf("%-16s T' -> $\n", cursor);
        return SUCCESS;
    }
}
 int F()
{
    if (*cursor == '(') {
        printf("%-16s F -> ( E )\n", cursor);
        cursor++;
        if (E()) {
            if (*cursor == ')') {
                cursor++;
```

```
            return SUCCESS;
        } else
            return FAILED;
    } else
        return FAILED;
} else if (*cursor == 'i') {
    cursor++;
    printf("%-16s F ->i\n", cursor);
    return SUCCESS;
} else
    return FAILED;
}
```

Output:


Enter the string
Input Action
i+(i+i)*i E -> T E'
i+(i+i)*i T -> F T'
+(i+i)*i F ->i
+(i+i)*i T' -> $
+(i+i)*i E' -> + T E'
(i+i)*i T -> F T'
(i+i)*i F -> (E)
i+i)*i E -> T E'
i+i)*i T -> F T'
+i)*i F ->i
+i)*i T' -> $
+i)*i E' -> + T E'
i)*i T -> F T'
)*i F ->i
)*i T' -> $
)*i E' -> $
*i T' -> * F T'
F ->i
T' -> $
E' -> $
String is successfully parsed
(base) niyatigupta@Maccy 2020BCSE073 %